KERNEL-LEVEL METHOD OF FLAGGING PROBLEMS IN APPLICATIONS

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Patent Application No. 60/486,638 entitled KERNEL-LEVEL METHOD OF FLAGGING PROBLEMS IN APPLICATIONS filed on July 11, 2003.

TECHNICAL FIELD

This application relates to flagging applications that may have memory leak and other resource usage problems using data collected at the kernel level.

BACKGROUND

Many currently available debugging tools require that programs be compiled and run in a different environment than their normal runtime environment, for example, with instrumented libraries and modules specifically used for debugging purposes. In addition, most debugging tools require many manual steps and time-consuming manual analysis in order to determine problems in an application. A hands-off tool that identifies potential problems with no intrusion to the applications while they are running in their natural run-time environment is desirable.

SUMMARY

A method of identifying potential problems in applications is provided. The method in one embodiment comprises monitoring at a kernel level system resource usage of one or more running applications without modifying

run-time environments of the running applications and identifying from the monitored system usage, an application whose system usage pattern satisfies a predetermined criteria associated with one or more problems. The system resource usage may include, but is not limited to memory usage.

A system for identifying problems in applications in one embodiment comprises a data collection module and a data analysis module. The data collection module in one aspect is operable to retrieve information about a running application at a kernel level. The data analysis module in one aspect is operable to determine from the retrieved information one or more system usage patterns that may be associated with one or more problems.

Further features as well as the structure and operation of various embodiments are described in detail below with reference to the accompanying drawings. In the drawings, like reference numbers indicate identical or functionally similar elements.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram illustrating a system kernel and various modules including the problem identifying module of the present disclosure in one embodiment.

Fig. 2 illustrates a flow diagram for monitoring kernel level system resource usage and collecting the related information.

Fig. 3 illustrates a flow diagram for analyzing the kernel level information.

DETAILED DESCRIPTION

Fig. 1 is a block diagram illustrating a system kernel and various modules including the problem identifying module of the present disclosure in one embodiment. The block diagram shown in Fig. 1 is presented for illustrative purposes only. Other Unix systems may have kernel models that deviate from the one shown in Fig. 1. Further, the problem identifying module of the present disclosure is not limited to applications in Unix Systems only, but rather, may be applied to other systems such as Windows, NT, Linux. Fig. 1 shows two levels: user 102 and kernel 104. The system call interface 106 represents the border between user programs 108a, 108b and the kernel. Libraries 110 map system calls invoked by the user programs 108a, 108b to the primitives needed to enter the kernel level 104. Assembly language programs may invoke system calls directly without a system call library. The libraries 110 are linked with the user programs 108a, 108b at compile time and thus may be considered as part of the user programs 108a, 108b.

The kernel level shown in Fig. 1 is partitioned into two subsystems: file subsystem 112 and process control subsystem 118. The file subsystem 112 accesses file data, for example, using buffering mechanism 114 that regulates data flow between the operating system and secondary storage devices via device drivers 116. The process control subsystem 118 is responsible for process synchronization 119, interprocess communication 120, memory management 122, and process scheduling 124. The file subsystem 112 and the process control subsystem 118 interact, for example, when loading a file into memory for execution. The process control subsystem 118 reads executable files into memory before executing them. The memory management module 124 controls the allocation of

memory. The scheduler module 122 allocates the CPU (central processing unit) to processes. The system interprocess communications 120 include asynchronous signaling of events and synchronous transmission of messages between processes. The hardware control 126 is responsible for handling interrupts and for communicating with the machine.

A process is the execution of a program and comprises a pattern of bytes that the CPU interprets as machine instructions (text), data, and stack. Several processes 128a, 128b or 130a, 130b may be instances of a user program 108a or 108b. The kernel loads an executable file into memory during, for example, an exec system call, and the loaded process comprises of at least three parts called regions: text, data, and the stack. A running process, thus, in one embodiment uses separate memory areas, allocating and deallocating the memory areas as the process runs. That is, once a process performs its desired functions, allocated memory is freed for other use. Similarly, a user program 108a or 108b may spawn one or more children processes and those children processes may in turn spawn additional processes to perform various tasks. Once the tasks are performed, however, under normal conditions, those processes should exit, either gracefully or with error conditions.

The data collection module 132 in one embodiment collects data related to the user programs 108a, 108b while allowing the user programs 108a, 108b to run in their natural run-time environment or mode, for example, without having to be recompiled or relinked to be run in a debug mode. The data collected by the data collection module 132, for example, may include, but are not limited to,

information about the memory such as the text, data, and stack memory used by the user programs 108a, 108b, the number of running or created processes per user programs 108a, 108b, the CPU usage per user programs 108a, 108b, and any other kernel or system resource usage data related to the user programs 108a, 108b.

The data collection module 132 in one embodiment utilizes exiting debugging tools to obtain the selected data. For example, the Q4 debugger released with Hewlett Packard's operating system, Linux kernel debuggers KDB or GDB, or the modular debugger (MDB) for Solaris may be utilized to retrieve information about the kernel space data and the various state of the system while the user programs 108a, 108b are running. An example of a Q4 script for retrieving such information may include calling Q4 as q4 -p /stand/vmunix /dev/mem, then invoking a script (such as a Perl script) that drives per-process data collection and saves selected fields of the output in a human readable format.

Similarly, other debugging tools may be used to generate a script for programmatically retrieving various system information at a kernel level. The retrieved information may be stored, for example, as a log file 138.

The data analysis module 134 may identify user programs that may have problems such as memory leaks, orphan processes by analyzing the information stored in the log file 138. The analysis may involve filtering the information and selecting those that meet predetermined criteria as described in more detail with reference to Fig. 3.

Fig. 2 illustrates a flow diagram for monitoring kernel level system resource usage and collecting the

related information. The monitoring and collecting, for example, may be performed periodically by executing existing kernel debugger facilities and extracting selected fields of data from the kernel debugger output. At 202, an interval period for monitoring is obtained, for instance, from a command line, from a file, or as an input to a query, or any other method for receiving input information. The interval, for example, may be every 600 seconds or the like. In another embodiment, the interval period may be obtained using starting and end time for monitoring. For example, a user may specify begin and end time and a number of times to monitor during that period.

At 204, processes that are running in a system are determined, for example, by calling subroutine LoadAllProcs. At 206, system resource statistics is obtained for each process running. The system resource statistics may be obtained by using existing facilities such as the Q4, MDB, or any other appropriate kernel debugging utilities, e.g., gdb /boot/vmlinux /dev/mem. For example, using the Q4 utility, memory page count allocated and used by a running process may be obtained and logged. The memory page count may be further typed into different types of memory being used by the running process: for example, text, data, and stack types. At 208, this information may be saved in a log file. The log file, for instance, may contain lines such as:

(PID)	(TEXT)	(DATA)	(STACK)
0x001	3	5	2
0x002	5	3	6
.	.	.	.

.	.	.	.
.	.	.	.
0x003	4	9	9
0x001	2	7	2
0x003	5	11	7

Similarly, a number of processes associated with a user program may be obtained to determine if any of the processes are defunct or orphaned by using respectively the process status or parent pid field as a selection criterion.

At 210, the method waits for the amount of time equivalent to the interval period, for example, by a sleep call for that amount of time. At 212, it is determined as to whether more monitoring is to be performed. If the monitoring is to continue, the method returns to 204. Otherwise, the method stops, performing any cleanups such as releasing memory, closing files, and exiting any children processes gracefully.

Fig. 3 illustrates a flow diagram for analyzing the kernel level information. At 302, the file that contains the system resource usage information is opened. At 304, the information is read. A line of a file, for example, may contain a process identifier and the memory usage for that process as shown above. In another example, a line of a file may contain an application identifier and the number of processes that are spawned by the process. In yet another example, a line of a file may contain a process identifier and the CPU usage for that process. At 306, the information is used to determine whether an abnormal pattern or condition exists. An abnormal pattern may exist, for example, if one particular process's memory size

increases continuously from period to period; if one particular long-running process's memory size continually increases without any decrease in size during that number of interval periods; if a process is running even when a parent process is not (orphan process); if a process is continuously spawning new processes.

A continuous increase in memory size for a particular process, for example, may be detected if the memory size logged in the log file (138 Fig. 1) for that process has increased from one period to another period of monitoring. A comparison of a memory size from the previous period to the current period, for example, may determine whether there is an increase from one period to another period. A number of increases then may be stored in a buffer and compared with a predetermined number to determine if the increase is excessive. Picking the right query interval and duration of testing depends on the application and should be chosen appropriately (e.g., every 60 seconds).

An existence of orphan process is detected, for example, if no parent process for a particular process is found in the log file as one of the running processes. In another example, a process is detected as spawning unusually large number of children processes if a predetermined number of processes all have the same parent process identifiers. At 308, the application or the process identifier detected as meeting the abnormal pattern or condition is then saved, for example, in another file (140 Fig. 1). At 310, if there are more lines to read in the log file, the method continues to 304. Otherwise, at 312, the method stops.

The system and method of the present disclosure may be implemented and run on a general-purpose computer. The

system and method of the present disclosure, for example, may be utilized during development of application programs or when run at a customer's site, for example, to detect and identify possible problems associated with the application programs. The embodiments described above are illustrative examples and it should not be construed that the present invention is limited to these particular embodiments. Although the description was provided using the Unix system as an example, it should be understood that the method and system disclosed in the present application may apply to any other computer operating systems. Thus, various changes and modifications may be effected by one skilled in the art without departing from the spirit or scope of the invention as defined in the appended claims.